Hardware Locks with Priority Ceiling Emulation for a Java Chip-Multiprocessor

Tórur Biskopstø Strøm
Department of Applied Mathematics
and Computer Science
Technical University of Denmark
torur.strom@gmail.com

Martin Schoeberl
Department of Applied Mathematics
and Computer Science
Technical University of Denmark
masca@dtu.dk

***Abstract*—According to the safety-critical Java specification, priority ceiling emulation is a requirement for implementations, as it has preferable properties, such as avoiding priority inversion and being deadlock free on uni-core systems.**

In this paper we explore our hardware supported implementation of priority ceiling emulation on the multicore Java optimized processor, and compare it to the existing hardware locks on the Java optimized processor.

We find that the additional overhead for priority ceiling emulation on a multicore processor is several times higher than simpler, non-premptive locks, mainly due to slow access to shared memory. We also find that PCE is mostly viable with large critical sections.

I. Introduction

Priority ceiling emulation (PCE) is a popular locking protocol for real-time systems. It allows preemption whilst preventing priority inversion and for uni-core systems it also prevents deadlocks. The safety-critical Java specification (SCJ) [1] requires that all implementations support PCE. As we target SCJ with our Java chip-multiprocessor, we explore the implementation of PCE in this paper. However, the protocol is more complex on multicore systems, so the theoretical benefits of the protocol might not be viable in practice.

The current draft of the SCJ specification [1] (p. 143) specifies that locks shared between threads running on different cores need to have a priority so that synchronized methods execute non-preemptively.

On a multicore system locking is usually implemented with the help of a compare-and-set instruction that operates atomically. However, supporting this instruction in a time-predictable way for shared memory becomes expensive; the worst-case execution time (WCET) for this instruction will be high. Therefore, direct on-chip hardware support for locking avoids the high execution time of external memory access.

We explored hardware support for locking in a Java multicore processor [2] with a single global lock and a locking unit in parallel to the single global lock [3]. From that starting point we integrated those two components into the Java locking unit (JLU). We use this JLU as a basis for our implementation of PCE on the JOP multicore processor. We analyze the worst-case execution time (WCET) of our PCE implementation and compare it to that of the global lock and JLU without PCE. We also compare benchmark results for the three locks.

The paper is organized as follows: Section II gives background information on locking and Section III presents the Java locking unit. Section IV describes our implementation of priority ceiling emulation and the necessary changes to the hardware locks. Section V analyses the implementation performance. Section VI presents related work. Section VII concludes the paper.

II. Priority Ceiling Emulation

When two or more computing routines share a stateful resource, and modification of the state is not implicitly atomic, it is necessary to ensure atomicity by other means. A common approach is to use synchronization locks, however these are subject to deadlocks and priority inversion.

There exist several protocols to alleviate the problems of deadlocks and priority inversion. Lui Sha et al. [4] describe the original priority ceiling protocol, which aims to prevent priority inversion and deadlocks. Under this protocol each lock has a priority assigned that is at least as high as the highest priority of any thread accessing it. If a thread owns a lock and another thread tries to acquire the same lock, the owning thread's priority is temporarily raised to that of the lock, preventing the other thread from executing until the owner has released the lock.

A simpler and, at least in our experience with uni-core systems, more common variation of this protocol is priority ceiling emulation [5] (PCE). With PCE a system raises a thread's priority to the lock's ceiling priority as soon as the thread acquires the lock, instead of waiting until there is contention. Other threads on the same core that would acquire the same lock are thereby prevented from executing until the owning thread has released the lock.

Although locking is well understood on uni-core systems, this is not the case for multicore systems, which schedule threads according to partitions. A single global partition allows all threads to execute on all cores, whereas a fully partitioned system fixes threads to specific cores. There also exist systems that mix the two partitioning methods.

Some of the benefits that locking protocols can provide on uni-core systems disappear when using multicore. Such is the case with dead-locks, where two threads on different cores

1555-0885/15 $31.00 © 2015 IEEE
DOI 10.1109/ISORC.2015.33

IEEE computer society

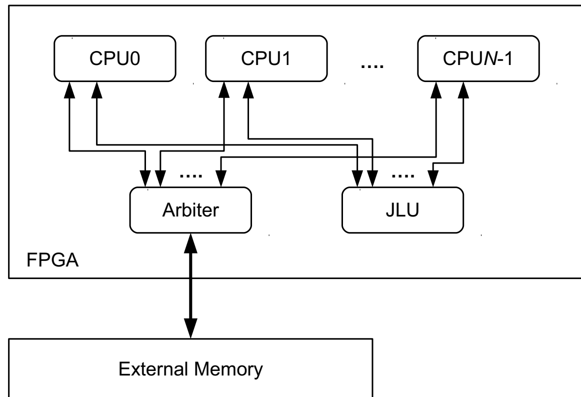

Fig. 1. A JOP CMP system with the JLU

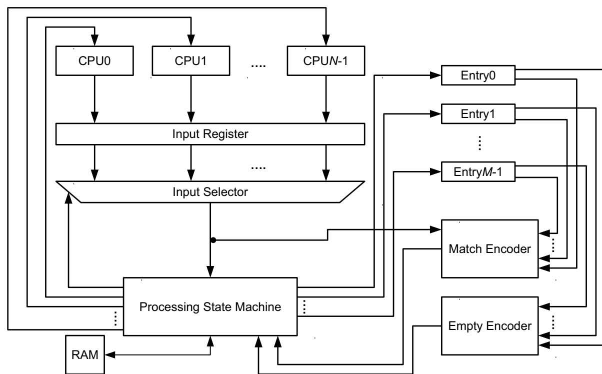

Fig. 2. The Java Locking Unit

can acquire two locks in opposite order, as the priority of each thread has no meaning to the opposing core.

Note that PCE still prevents priority inversion on multicore systems, as long as resource sharing between scheduling domains is non-preemptive, which is the case with SCJ. Sharing locks across partitions does not enable lower priority threads within the same core(s) from preempting higher priority threads, which is the requirement to prevent priority inversion.

III. The Java Locking Unit

We explore the PCE multicore locking and the hardware support for it on the Java processor JOP [6]. We have chosen JOP because the hardware is open source and relatively easy to extend. The runtime of JOP also includes a first prototype of SCJ level 0 and level 1 [7]. Furthermore, a chip-multiprocessor (CMP) version of JOP is available [2]. Note that the hardware support for locks is not JOP specific and possibly even usable in non-Java multicore systems.

Figure 1 shows our configuration. Several JOP cores connect to the shared memory via an arbiter. In addition to the arbiter, there is the Java locking unit (JLU). Figure 2 shows an overview of the JLU's structure. The JLU maps into the I/O space for each core and contains a set of registers that represent lock "entries". Each entry consists of a flag indicating whether the entry is empty, a word (32-bit in the case of JOP) with a lock's address, and a counter to track the number of times a core acquires the lock. When requesting a lock, a core writes a lock's address to the unit and the unit checks if the lock's address already exists in the entries. If so, another core already owns the lock and the JLU enqueues the current core in a FIFO residing in a local on-chip memory. Otherwise the JLU registers the lock in an empty entry and registers the requesting core as the owner. Requests are handled in round-robin order.

Before requesting a lock, a thread disables its core's interrupts. In addition, during the JLU's processing of either an acquisition or release, the JLU blocks the core's write request on the interconnect. This stalls the core until the request is complete and the core has acquired/released the lock, effectively making the thread spin-wait at top priority. If the core releases its final lock the thread enables interrupts.

Requesting a lock in the JLU has a low overhead. While waiting for a lock, a thread non-preemptively spin-waits. While owning a lock, a thread non-preemptively executes the critical section. The JLU locking can therefore be considered a degenerated form of PCE, hereafter referred to as DPCE. The main difference between PCE and DPCE is that PCE allows preemption when there are locks with ceiling priorities that are lower than some thread priorities.

According to the SCJ specification, critical sections, protected by locks that are shared between cores, have to run at top priority, as priorities between scheduling partitions have no meaning. This means that DPCE behaves like PCE except when a lock is not shared between cores and has a ceiling priority that is lower than at least one other thread on the core that does not access the lock.

IV. PCE Implementation

As in standard Java, SCJ allows each object to serve as a lock. In addition, SCJ allows the ceiling priority of each lock to be configurable. It is therefore necessary to register and remember the priority of object's throughout their lifetime. However, SCJ requires that each lock priority not explicitly set uses a system default priority, so PCE implementations need only register an object's priority when that priority is other than the default.

There are multiple options to maintain a object/priority mapping, e.g., a hash map, a table, the object header, etc. As a quick and efficient solution we found unused space in JOP's object header alongside the scope levels. This header field is 32 bits long, which we split into two 16 bit words, with one maintaining scope levels and the other maintaining an object's (potential) ceiling priority. This limits both the scopes and the priorities to 65536 levels. However, we find this to be more than adequate for most, if not all, solutions. If the priority field is zero, the ceiling of the object is not set, so if a thread uses the object as a lock, the ceiling will be the system default.

The JLU implements DPCE, meaning that threads always spin wait at top priority. It is therefore necessary to make some

changes to the JLU before it supports proper PCE.

The JLU only maintains queues of cores waiting for locks and therefore doesn't distinguish between threads on each core. It is therefore an issue if two threads on the same core try to request the same lock, as the JLU would only enqueue the core once. However, from PCE we know that threads requesting the same lock should not have a priority higher than the ceiling of the lock. Therefore, other threads on the same core trying to acquire the same lock will not be scheduled as long as one of the threads is holding the lock.

To support preemption, we have modified the JLU's locking procedure. Instead of disabling interrupts until a core releases all its locks, interrupts are only disabled for the duration of a request, i.e., the core immediately owns the lock, the JLU has enqueued the core or the core released the lock. Furthermore, we add a request port to the JLU that returns the state of a lock, i.e., whether the current core is the owner or not. A thread can thereby spin in software (preemptively) while checking if it has become the owner.

One of the issues with PCE is the necessity to track priorities as a thread acquires different locks, so that the thread gets the priorities in reverse order as it releases the locks. To support this functionality we implemented priority "bread crumbs". Each thread contains an array index as well as a priority array and a lock count array, both with length $k + 2$, where k is the number of locks with a priority other than the default. k is found by registering all ceiling modifications and counting the number of different priorities.

V. Evaluation

For the evaluation we compare the PCE implementation with JOP's multicore global lock and also with the unmodified hardware locks (the JLU) on the Altera DE2-70 board.

The number of lock entries in the hardware unit is configurable, but we used the default configuration with 32 entries. Previous tests have shown that the number of entries does not, or minimally, affect the performance and only affect the size of the hardware.

Our performance analysis consists of 2 parts: WCET analysis of the locking routines using JOP's WCET tool, and benchmarks using JemBench [8].

Table I shows the WCET analysis results for the different locking routines. For the global lock and JLU analysis we manually counted the number of microcode steps for `monitorenter` and `monitorexit`. We also counted the number of hardware cycles used by the global lock and JLU by analyzing the hardware state machines. For the PCE analysis we similarly had to analyze microcode, as well as counting cycles in the statemachine. Additionally, we had to analyze the two software routines, `f_monitorenter` and `f_monitorexit`, that read, track and update the priorities.

The results show that the additional complexity of PCE has quite a negative impact on the locking/unlocking performance. The WCET increase relative to the core count is understandable, as some of the software steps have to access shared memory and therefore have to go through the arbiter. With a WCET that is at least 20 times higher than the JLU or global lock, a theoretically schedulable task set with PCE might not be practically schedulable.

TABLE I
WCET in clock cycles for each locking unit and its corresponding locking routine

	Cores			
	1	2	4	8
`monitorenter`				
Global Lock	19	19	19	19
JLU	28	33	43	63
PCE	615	876	1382	2374
`monitorexit`				
Global Lock	20	20	20	20
JLU	23	28	38	58
PCE	570	801	1255	2163

TABLE II
Benchmark results, iterations/second

	Cores			
	1	2	4	8
NQueens N=9, L=3				
Global lock	29	57	103	143
JLU	29	56	102	143
PCE	28	53	96	125
AES N=4, L=N/A				
Global lock	87	140	127	89
JLU	87	139	127	89
PCE	86	138	126	88
Increment N=48, L=1				
Global lock	34	28	29	16
JLU	34	31	52	66
PCE	26	24	41	55
Increment N=48,L=100				
Global lock	0.79	0.48	0.35	0.22
JLU	0.79	0.72	1.22	1.52
PCE	0.78	0.72	1.21	1.52

Table II shows the benchmarking results. N is the number of runnables generated in the benchmark and L is the benchmark specific load. We only included benchmarks that actually used locks. In addition, **Increment** is not part of the standard JemBench suite, but added by us to have a benchmark that heavily relies on locking. In **Increment**, every runnable locks on each runnable and increments its counter L times. A high L therefore corresponds to a large critical section.

The results show that PCE is worse than both the JLU and the global lock, when there is little contention, such as for **NQueens** and **AES**. However, PCE is better than the global lock in **Increment** where there is heavy contention. In addition, whereas the non-preemptive JLU is generally preferred, PCE becomes a viable alternative when the critical sections become large and PCE's overhead is comparatively low, as seen for **Increment** $L = 100$.

VI. Related Work

We are not the first to implement PCE for SCJ. The Hardware near Virtual Machine [9] is a uni-core platform that implements SCJ and PCE. When an application sets the priority of an object, the system creates a monitor object and sets a reference to it in the lock object's header. The monitor object contains the specified priority and a field for the acquiring handler's priority. When a handler acquires the lock the system saves its priority in the monitor and the handler gets the monitor's priority (if it is higher). When the handler releases the lock, the handler regains its old priority. This means that a handler can acquire locks with different priorities, but when releasing a higher priority lock it will regain the priority of the last lock. We have the same support in our implementation, although our solution differs somewhat, as explained in Section IV. In the future we might choose to implement the monitor object solution, as this is more elegant. However, this is not enough on multicore systems, where the system needs to maintain a queue for threads on other cores waiting for the same lock.

Yodaiken [10] argues against priority inheritance on the grounds that it adds complexity and is inefficient. This is similar to our issue with PCE, although we admit that the issues can be far more prominent in priority inheritance than in PCE.

Yodaiken also argues that instead of using priority inheritance to solve the issue of priority inversion, the solution is either to: (1) Make the resource related operations atomic and fast, or (2) remove the contention or (3) priority schedule the operations. With regards to (1) the author argues that RTLinux programmers use the `pthread_spin_lock` operation to disabled interrupts and, in the case of multicore systems, let threads spin while waiting for a lock. This solution is equivalent to the DPCE behavior of non-preemptive spinning.

Brandenburg et al. [11] extend the LInux Testbed for MUltiprocessor Scheduling in Real-Time systems (LITMUS^{RT}) with resource sharing and then empirically evaluate lock-free execution, wait-free execution, spin-based locking, and suspension-based locking. They do this under the Flexible Multiprocessor Locking Protocol (FMLP). They conclude that systems should avoid suspension when threads share resources across partitions. This supports the SCJ specification, which requires that all locks shared across partitions should lock non-preemptively. However, they also conclude that suspending is never preferable to spinning, and this will most likely always be the case unless a system spends at least 20% of its time in critical sections. In our paper we analyze DPCE and PCE, used for spinning and suspending respectively. We similarly argue that the simplicity of DPCE can render the theoretical benefits of PCE void when one adds the actual locking overhead to the analysis.

VII. Conclusion

We have described our implementation of priority ceiling emulation with hardware support. The added complexity of supporting priority modifications on locking adds as much as 20 times the number of cycles to the overall locking performance compared to only using the hardware locks. This can result in a theoretically schedulable task set being practically un-schedulable when switching to priority ceiling emulation. We also find that PCE is mostly usable with large critical sections which offset PCE's overhead.

Source Access

Our implementation of PCE is open source and available at https://github.com/torurstrom/jop.git on the `pce` branch.

Acknowledgments

This work was partially funded by the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST).

References

[1] D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. O. Nielsen, K. Nilsen, M. Schoeberl, J. Vitek, and A. Wellings, *Safety-Critical Java Technology Specification, Draft*, Java Community Process Std., 2014. [Online]. Available: https://github.com/scj-devel/doc/blob/master/scj-0-100.pdf

[2] C. Pitter and M. Schoeberl, "A real-time Java chip-multiprocessor," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 1, pp. 9:1–34, 2010. [Online]. Available: http://www.jopdesign.com/doc/jopcmp_tecs.pdf

[3] T. B. Strøm, W. Puffitsch, and M. Schoeberl, "Chip-multiprocessor hardware locks for safety-critical Java," in *Proceedings of the 11th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2013)*. Karlsruhe, DE: ACM, October 2013, pp. 38–46. [Online]. Available: http://www.jopdesign.com/doc/cmphwlocks.pdf

[4] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *Computers, IEEE Transactions on*, vol. 39, no. 9, pp. 1175–1185, 1990.

[5] A. Burns and A. J. Wellings, *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*, 3rd ed. Addison-Wesley Longman Publishing Co., Inc., 2001.

[6] M. Schoeberl, "A Java processor architecture for embedded real-time systems," *Journal of Systems Architecture*, vol. 54/1–2, pp. 265–286, 2008. [Online]. Available: http://www.jopdesign.com/doc/rtarch.pdf

[7] M. Schoeberl and J. R. Rios, "Safety-critical Java on a Java processor," in *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*. Copenhagen, DK: ACM, October 2012, pp. 54–61. [Online]. Available: http://www.jopdesign.com/doc/jopscj.pdf

[8] M. Schoeberl, T. B. Preusser, and S. Uhrig, "The embedded Java benchmark suite JemBench," in *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010)*. New York, NY, USA: ACM, August 2010, pp. 120–127. [Online]. Available: http://www.jopdesign.com/doc/jembench.pdf

[9] H. Søndergaard, S. E. Korsholm, and A. P. Ravn, "Safety-critical Java for low-end embedded platforms," in *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*. Copenhagen, DK: ACM, October 2012.

[10] V. Yodaiken, "Against priority inheritance," 2004.

[11] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. H. Anderson, "Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin?" in *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE*. IEEE, 2008, pp. 342–353.